19/01/2024
Friday

* **Partition Equal Subset Sum (leetcode-416)**

Given array nums, return true if you can partition the array into two subsets such that the sum of the elements in both subsets is equal or false otherwise.

I/P- nums = [1,5,11,5]

O/P- true

Explanation- The array can be partitioned as [1,5,5] and [11].

Approach-

= If sum of the elements in nums is odd, it means it cannot be partitioned into two subsets.

= If sum of the elements in nums is even, also if there exists a subset of nums array whose sum is half of the sum of the elements of nums array, it means, it can be partitioned into two subsets.

- Code using recursion-

```
class Solution{
public:

bool solveUsingRec(vector<int>&nums, int index,
                   int targetSum){
    //base cases
    if(targetsum==0) return 1;
    if(targetSum<0 || index >= nums.size()){
        return 0;
    }

    bool include= solveUsingRec(nums, index+1,
                                targetSum-nums[index]);
    bool exclude= solveUsingRec(nums,index+1,targetSum);

    return (include||exclude);
}

bool canPartition(vector<int>& nums){

    int sum=0;
    for(auto num:nums){
        sum+=num;
    }
    // if odd
    if(sum&1) return 0;

    // if even
    return solveUsingRec(nums, 0, sum/2);
}
};
```

- Code using memoization-

```
class Solution{
public:

bool solveUsingMemo (vector<int> &nums , int index,
 int targetSum, vector<vector<int>>& dp) {
    //base cases
    if (targetSum == 0) return 1;
    if (targetSum < 0 || index >= nums.size()) {
        return 0;
    }
    if (dp[index][targetSum] != -1) {
        return dp[index][targetSum];
    }
bool include= solveUsingMemo (nums, index+1,
                              targetSum- nums [index]);
bool exclude = solveUsingMemo ( nums,index+1,targetSum);

return dp[index][targetSum]= (include ||exclude);
}
bool canPartition (vector<int> &nums){
    int sum= 0;
    for (auto num: nums) {
        sum += num;
    }
    if (sum & 1) return 0;

vector<vector<int>> dp (nums.size()+1,vector<int>(sum/2+1,-1

return solveUsingMemo (nums, 0, sum/2, dp);
}
};
```

- Code using tabulation –

```
class Solution {
public:

bool solveUsingTabu(vector<int>&nums, int targetSum){
    int n= nums.size();
    vector<vector<int>>dp(n+1,vector<int>(targetSum+1,0));

// if targetSum 0 ho jaye whatever be the index, answer
// 1 hoga –
    for(int row=0; row<=n; row++){
        dp[row][0]=1;
    }
for(int i=n-1; i>=0; i--){
  for(int j= 0; j<=targetSum; j++){
    bool include= 0;
    if(j-nums[i]>=0){
      include= dp[i+1][j-nums[i]];
    }
    bool exclude= dp[i+1][j];

    dp[i][j]= (include||exclude);
  }
}
  return dp[0][targetSum];
}
bool canPartition(vector<int>&nums){
    int sum=0;
    for(auto num:nums) sum+=num;
    if(sum&1) return 0;
    return solveUsingTabu(nums, sum/2);
  }
};
```

Optimising the tabulation code space-wise →

dp[i][j] → dp[i+1][j-nums[i]]
dp[i][j] → dp[i+1][j]

By using only two rows or vectors, we can do this as the answer of current cell depends upon the answer of next rows cells only.

```
class Solution{
public:

bool solveUsingTabuSO(vector<int>& nums, int targetSum){
    int n= nums.size();
    vector<int> next(targetSum+1, 0);
    vector<int> curr(targetSum+1, 0);

// for the targetSum = 0
    curr[0] = 1; // because if targetsum gets 0
    next[0] = 1; // it means subset exists

for(int i=n-1; i>=0; i--){
 for(int j=0; j<=targetSum; j++){
   bool include=0;
   if(j-nums[i]>=0){
     include= next[j-nums[i]];
   }
   bool exclude= next[j];

   curr[j] = (include || exclude);
  } next = curr; //shifting
 } return next[targetSum];
}
bool canPartition(vector<int>& nums){
 int sum=0;
 for(auto num: nums) sum+=num;
 if(sum&1) return 0;
 return solveUsingTabuSO(nums, sum/2);
}
};
```

* Number of dice rolls with target sum -

Given n dices, each dice has k faces numbered from 1 to k. Given a target, return the number of possible ways to roll the dice, so that sum of face-up numbers equals target.

I/P - n=2, k=6, target=7

O/P - 6

Explanation - There are 6 ways to get sum=7 →
{1,6}, {2,5}, {3,4}, {4,3}, {5,2}, {6,1}

Approach -

We are using here explore all ways pattern. for 2 dices and 6 faces each, dice1 has 6 choices and for every choice of dice 1, dice 2 has 6 choices.
So there are 6x6 possible choices out of which 6 are there with the target sum.

- Code using recursion -

```
class Solution{
public:
  int MOD = 1000000007;
  int solveUsingRec (int n, int k, int target, int diceUsed,
                     int currSum){

    //base cases
    if (diceUsed==n && currSum==target) return 1;
    if (diceUsed!=n && currSum==target ||
        diceUsed==n && currSum!=target) return 0;

    int ans=0;
    for (int face=1; face<=k; face++){
      int recAns=0;
      if (currSum+face<=target){
        recAns = solveUsingRec (n, k, target, diceUsed+1,
                                currSum+face) % MOD;
        ans = (ans %MOD + recAns %MOD) %MOD;
      }
    return ans;
  }

int numRollsToTarget (int n, int k, int target){
    int diceUsed=0;
    int currSum=0;

    return solveUsingRec(n,k,target,diceUsed,currSum);
  }
};
```

- Code using memoization -

```
class Solution{
public:
int MOD= 1000000007;

int solveUsingMemo(int n, int k, int target, int diceUsed,
vector<vector<int>>&dp, int currSum){

    //base cases
    if (diceUsed==n && currSum==target){
            return 1;
    }
    if(diceUsed==n && currSum!=target || diceUsed!=n &&
      currSum==target) return 0;
    if (dp[diceUsed][currSum] != -1) return dp[diceUsed][currSum];

    int ans=0;
    for(int face=1;face<=k;face++){
      int recAns=0;
      if(currSum+face<=target){
      recAns= solveUsingMemo(n,k,target,diceUsed+1,dp
                             currSum+face)%MOD;
      ans= (ans%MOD + recAns%MOD)%MOD;
    }
    return dp[diceUsed][currsum]=ans;
}
int numRollsToTarget(int n,int k,int target){
      int diceUsed=0;
      int currSum=0;
vector<vector<int>>dp(n+1,vector<int>(target+1,-1));
return solveUsingMemo(n,k,target,diceUsed,dp,currSum);
}
};
```

- Code using tabulation -

```
class Solution {
public:

int solveUsingTabu ( int n, int k, int target) {
 vector<vector<int>> dp (n+1, vector<int>(target+1, 0));

  dp[n][target] = 1;

 for (int diceUsed=n-1; diceUsed>=0; diceUsed--){
  for (int currSum=target; currSum>=0; currSum--){
     int ans=0;
    for (int face=1; face<=k; face++) {
      int recAns=0;
      if (currSum + face <=target){
        recAns = dp[diceUsed+1] [currSum+face];
      }
      ans=(ans %MOD + recAns %MOD) %MOD;
    }
    dp [diceUsed][currSum] =ans;
  }
 }
  return dp [0][0];
}
int numRollsToTarget ( int n, int k, int target) {
      int diceUsed= 0;
      int currSum= 0;

      return solveUsingTabu (n, k, target);
 }
};
```

dp[diceUsed][currSum] ⟶ dp[diceUsed+1][currSum+face]

- **Space optimisation in tabulation code-**

```
class Solution{
public:
int MOD = 1000000007;

int solveUsingTabuSO(int n, int k, int target){

    vector<int> next(target+1, 0);
    vector<int> curr(target+1, 0);

    next[target] = 1;
 for(int diceUsed = n-1; diceUsed >= 0; diceUsed--){
  for(int currSum = target; currSum >= 0; currSum--){
     int ans = 0;
     for(int face = 1; face <= k; face++){
        int recAns = 0;
        if(currSum + face <= target){
           recAns = next[currSum + face]; }
           ans = (ans % MOD + recAns % MOD) % MOD;
     }
     curr[currSum] = ans;
  }
  //shifting
   next = curr;
 }
  return next[0];
}
int numRollsToTarget(int n, int k, int target){
    int diceUsed = 0;
    int currSum = 0;
    return solveUsingTabuSO(n, k, target);
  }
};
```